Using Historical Information for Fuzzing JavaScript Engines

Bruno Gonçalves de Oliveira[1][a], Andre Takeshi Endo[2][b] and Silvia Regina Vergilio[1][c]
[1]*Department of Computer Science, Federal University of Paraná, Curitiba, PR, Brazil*
[2]*Computing Department, Federal University of São Carlos, São Carlos, SP, Brazil*

Keywords: Fuzzing, JavaScript Engine, Security, Vulnerabilities, Exploits.

Abstract: JavaScript is a programming language commonly used to add interactivity and dynamic functionality to websites. It is a high-level, dynamically-typed language, well-suited for building complex, client-side applications and supporting server-side development. JavaScript engines are responsible for executing JavaScript code and are a significant target for attackers who want to exploit vulnerabilities in web applications. A popular approach adopted to discover vulnerabilities in JavaScript is fuzzing, which involves generating and executing large volumes of tests in an automated manner. Most fuzzing tools are guided by code coverage but they usually treat the code parts equally, without prioritizing any code area. In this work, we propose a novel fuzzing approach, namely JSTargetFuzzer, designed to assess JavaScript engines by targeting specific source code files. It leverages historical information from past security-related commits to guide the input generation in the fuzzing process, focusing on code areas more prone to security issues. Our results provide evidence that JSTargetFuzzer hits these specific areas from 3.61% to 16.17% more than a state-of-the-art fuzzer, and covers from 1.46% to 15.09% more branches. By the end, JSTargetFuzzer also uncovered one vulnerability not found by the baseline approach within the same time frame.

1 INTRODUCTION

JavaScript is a predominant programming language on the Web, commonly used to add interactivity and dynamic functionality to websites. It is a high-level, dynamically-typed language, well-suited for building complex, client-side applications and supporting server-side development. Most websites use Javascript code to implement different tasks in specific areas, including back-end servers with approaches like Node.js[1], and front-end UIs like React.js[2]. One of the key advantages of JavaScript is its widespread availability, as it is supported by all modern web browsers, which can be used on various devices. This makes it a popular choice for building web applications accessible from any device with an Internet connection.

JavaScript is always supported by JavaScript engines in web browsers. The engines are embedded and can interpret and execute JavaScript code during Internet browsing. They generally share the same architecture, which includes parser, interpreter, baseline compiler, and *Just-In-Time (JIT)* compiler (or optimizer) (Kienle, 2010). While these features enhance performance, they also introduce unique or distinct vulnerabilities that are not typically found in other types of software (Park et al., 2020). The engines generally have particular security issues, from memory allocations and protections to improper data validation (Kang, 2021; Lee et al., 2020; Groß et al., 2023). The vulnerabilities stem directly from the dynamic nature of JIT compilation and optimization processes, which can create security gaps. A user could be deceived into executing malicious JavaScript code embedded within a web page's payload designed to exploit vulnerabilities in a JavaScript engine, potentially leading to remote command execution.

JavaScript engine is then an unquestionably sensitive piece of software for security reasons. A vulnerability in a JavaScript engine can have far-reaching consequences, including unauthorized data access, code execution, and privacy breaches. Therefore, ensuring the security of JavaScript engines is critical to protecting users and systems from potential attacks. Identifying and fixing vulnerabilities in JavaScript

[a] https://orcid.org/0009-0008-4554-5166
[b] https://orcid.org/0000-0002-8737-1749
[c] https://orcid.org/0000-0003-3139-6266
[1]https://nodejs.org
[2]https://reactjs.org

Gonçalves de Oliveira, B., Endo, A. T. and Vergilio, S. R.
Using Historical Information for Fuzzing JavaScript Engines.
DOI: 10.5220/0013417700003929
In *Proceedings of the 27th International Conference on Enterprise Information Systems (ICEIS 2025) - Volume 2*, pages 59-70
ISBN: 978-989-758-749-8; ISSN: 2184-4992

engines as quickly as possible is fundamental, and many researches have been conducted to help in this task (He et al., 2021; Tian et al., 2021). Various techniques and tools were proposed for testing and validation, code review and analysis[3], and use of security best practices[4] in the development process. One of these most popular techniques is fuzzing, which involves generating and executing large volumes of tests in an automated manner to discover vulnerabilities in the engine (Tian et al., 2021). Many fuzzing tools have been released for JavaScript engines (Han et al., 2019; Holler et al., 2012; Wang et al., 2019; Lee et al., 2020); they usually have different methods to solve the syntax and semantics problems during the input generation. The mutational strategy generates inputs based on an initial corpus of JavaScript code and the generational is based on the JavaScript grammar. Most fuzzing tools generally have static configuration and do not allow significant modifications in the fuzzing process, which may limit their range of found vulnerabilities.

Recent advances in the field emphasize the role of code coverage in enhancing fuzzing strategies. Code coverage metrics allow fuzzers to systematically explore the JavaScript engine, ensuring that various aspects of the codebase are tested thoroughly. This approach helps to reach unexplored parts of the program and evaluate alternative methods for input generation during fuzzing campaigns (Eom et al., 2024). Fuzzilli (Groß et al., 2023) is an example of coverage-oriented tool, and is considered the state-of-the-art fuzzing tool (Bernhard et al., 2022). It utilizes an *Intermediate Language (IL)* to build test inputs with valid syntax and semantics, which allows support for various JavaScript engines. The tool implements a mutational-based input generation with multiple operators defined separately. The algorithm implemented by Fuzzilli ensures that new input programs remain syntactically valid by evolving from previously successful samples. This process helps maintain the integrity of JavaScript inputs while efficiently exploring and covering new execution paths within the engine.

Despite its potential, the code coverage based technique is often underutilized. Existing tools (Han et al., 2019; Wang et al., 2019) usually rely on simplistic metrics that merely check if new code paths have been reached, without delving deeper into other characteristics of the covered code. Another limitation inherent to these tools is the time taken to execute the algorithms. They often do not provide features to be configured for the fuzzing process, taking a generic approach for all types of vulnerabilities (Holler et al., 2012; Lee et al., 2020; Han et al., 2019). Newer fuzzing tools target a unique type of vulnerability but do not provide resources for others (Sun et al., 2022; Bernhard et al., 2022). As a consequence, their effectiveness and ability to detect certain types of vulnerabilities are reduced.

In light of these limitations, this paper introduces a history-based approach, namely JSTargetFuzzer, which enables targeted fuzzing in JavaScript engines by leveraging historical data to guide the fuzzing process. Historical data refers to past information on security-related commits, bug reports and fixes, and previous patches within JavaScript engine repositories. This data offers a window into the evolutionary patterns of software systems, highlighting areas that have been repeatedly modified or inadequately patched and, thus, more likely to harbor vulnerabilities. This is particularly relevant in complex software systems, where residual vulnerabilities often persist after incomplete fixes and where patches can sometimes introduce new defects by focusing on these historically vulnerable code segments (Li and Paxson, 2017; Shin and Williams, 2008).

We implemented our approach on the top of Fuzilli (Groß et al., 2023). Our tool incorporating historical data from security-related JavaScript engine commits to guide the generation of programs or inputs during the fuzzing campaigns. Our results provide evidence that JSTargetFuzzer is capable of targeting a specific code area, leading to concentrating efforts and discovering more branches in the target code area than Fuzzilli. Within the time frame of the experiments, JSTargetFuzzer was also capable of uncovering one vulnerability missed by Fuzzilli.

This paper is organized as follows: Section 2 reviews related work; Section 3 introduces JSTargetFuzzer; Section 4 presents implementation details of our tool; Section 5 describes the experimental settings of the evaluation conducted; Section 6 analyses the obtained results; Section 7 discusses limitations and possible threats to the results validity; and Section 8 concludes the paper.

2 RELATED WORK

Fuzzing JavaScript engines has been an active research topic, receiving attention from practitioners and researchers. Jsfunfuzz (Mozilla, 2022) is probably the first public fuzz testing tool for JavaScript engine, developed for the JavaScript engine Mozilla's SpiderMonkey and released in 2007. The tool, even today, is a benchmark for fuzzing techniques since it

[3]https://www.sonarsource.com/

[4]https://www.ncsc.gov.uk/collection/developers-collection

demonstrated a great capability in discovering new vulnerabilities. LangFuzz (Holler et al., 2012) is a generic mutational fuzzing tool that can discover vulnerabilities in different programming languages, including JavaScript. Both tools struggle to generate semantically correct JavaScript code (Han et al., 2019).

To overcome this challenge, other fuzzers have been proposed in the literature. CodeAlchemist (Han et al., 2019) tries to solve semantics problems while generating test cases. It takes JavaScript source files as seeds and converts them to *Abstract Syntax Trees (ASTs)*. After that, it separates them into code blocks, comprehends their order, and generates new code snippets. Then, CodeAlchemist remounts the source files and utilizes them for fuzzing using their marked order. Superion (Wang et al., 2019), an extension of *American Fuzzy Lop (AFL)*[5], brings the grammar-aware capability to AFL, allowing the fuzzer to generate input data that conform to specific syntactic rules or grammar defined for the target program. Another research direction is to use Neural Network Language Models, like Montage (Lee et al., 2020), to support the fuzzing process. It converts seed files into ASTs, identifies their sequence, and use this information to train the models. More recently, CovRL (Eom et al., 2024) integrates coverage feedback directly into *Large Language Models (LLMs)*, along with a reinforcement learning algorithm to guide the mutation process.

Other fuzzers are designed to reveal a specific kind of vulnerability. For instance, KOP-Fuzzer (Sun et al., 2022) is a fuzzing tool designed to target Type Confusion vulnerabilities. There is also interest on vulnerabilities related to JIT optimizations in JavaScript engines (Park et al., 2020; Bernhard et al., 2022; Wang et al., 2023). DIE (Park et al., 2020) targets JIT vulnerabilities by maintaining two aspects of *Proof-of-Concepts (PoCs)* used as seeds: the preservation of data types and the structural integrity of the code during mutation. This approach ensures that the generated test cases closely mimic real-world scenarios where specific data types and structures can trigger vulnerabilities in JIT-compiled code. JIT-Picker (Bernhard et al., 2022) relies on differential fuzzing, testing the engine against itself, by running the program twice, with and without the JIT compiler enabled. Finally, FuzzJIT (Wang et al., 2023) implements a sophisticated template-based approach, ensuring that every generated program triggers the JIT optimization process.

Among the several initiatives for fuzzing JavaScript engines, one has been particularly successful. Fuzzilli (Groß et al., 2023) is a fuzzing tool developed by Google Project Zero[6] that targets JavaScript engines. The tool tries to solve the semantics problems by implementing an *Intermediate Language (IL)* for handling JavaScript source code. The IL is used to convert JavaScript code into a simplified form for manipulation. The tool also implements a guided technique by instrumenting the JavaScript engines to obtain code coverage. The algorithm implemented by Fuzzilli ensures that new input programs remain syntactically valid by evolving from previously successful samples. This process helps maintain the integrity of JavaScript programs while efficiently exploring new execution paths within the engine.

Fuzzers for different domains have previously explored historical code analysis to enhance their fuzzing strategies (Xiang et al., 2024; Zhu and Böhme, 2021).

However, to the best of our knowledge, no existing fuzzing tool utilizes historical information concerning security-related commits to guide the fuzzing process in JavaScript engines. This presents an opportunity to identify vulnerabilities by analyzing the repository history of the JavaScript engine, allowing prioritization of code areas more likely to contain security flaws.

3 PROPOSED APPROACH

This section introduces JSTargetFuzzer, an approach that utilizes historical data to focus fuzzing efforts on specific source code files within JavaScript engines. Historical information, including detailed records of past changes to the codebase, such as security patches, commit logs, and bug reports, can provide invaluable insights, and point out files and functions that have been repeatedly modified in response to vulnerabilities, highlighting areas of the code that are potentially more prone to security flaws.

The key insight is to maximize the chances of identifying vulnerabilities by covering security-sensitive areas, where future issues may arise, as the intrinsic complexity of software often means that parts of code that have previously exhibited vulnerabilities are likely to do so again. Incomplete fixes can leave residual vulnerabilities, and patches themselves can sometimes introduce new defects (Li and Paxson, 2017; Shin and Williams, 2008). Persistent security threats from incomplete fixes have been a notable concern for JavaScript. For instance, the incomplete

[5]https://github.com/google/AFL

[6]https://googleprojectzero.blogspot.com

resolution of CVE-2018-0776[7] in the Microsoft's engine ChakraCore resulted in the emergence of vulnerabilities CVE-2018-0933[8] and CVE-2018-0934[9]. Notably, these patches involved revisions to the same file.

Figure 1 gives an overview of our fuzzing approach. Different colors in the diagram are used: green to distinguish JavaScript engine elements, orange for the historical contribution, purple to the weighting system, and blue to the fuzzing process. We can also see two main paths. The first one (1) is related to historical information retrieval, and the second (2)is related to the adoption of retrieved information in the fuzzing process. To perform the first, the security professional selects the target JavaScript engine. Then the corresponding repository is mined, and security-related commits are found. From these commits, security-related files are identified and ranked. In the second path, the security professional configures the weighting system using the code coverage capability, which is informed by the ranking results. Ultimately, while focusing on specific files, the fuzzing process may uncover new vulnerabilities and generate additional test cases. The following sections describe the elements of the approach in detail.

3.1 Historical Information Retrieval

The goal of this trial is to obtain security-related historical information and rank the JavaScript files.

3.1.1 Mine Commits

Once the engine is provided, its corresponding commit history of the JavaScript engine is collected from its source code repository, such as GitHub[10]. The availability of the source code is crucial, as well as the access to the full commit history, for extracting and classifying the relevant commits. This foundational activity sets the stage for the subsequent phases of our approach, where we analyze commit messages and evaluate changes in the source code. Examining these commits provides valuable insights into the engine's evolution, particularly concerning security-related modifications. This information is instrumental in shaping the weighting system that guides the fuzzing process (see Section 3.2), allowing a focus on the most vulnerable and frequently altered areas of the codebase.

3.1.2 Identify Security-Related Commits

At this point, we need to classify the commits as security-related or not. This security-related classification implies that any vulnerability aspect is being handled within the commit. We can use different techniques to achieve this goal, such as manual identification, searching for common security-related keywords, and utilizing a *Machine Learning (ML)* classification model. In these cases, the classification is done by recovering commits' messages and titles and inspecting the texts' descriptions to recognize a security aspect.

3.1.3 Rank Security-Related Files

The process involves enumerating and ranking the files within the JavaScript engine frequently modified in previously identified security-related commits. Analyzing the frequency of changes to these files helps identify which parts of the codebase have been most impacted by security fixes. This ranking provides insight into the areas of the engine that are likely to contain vulnerabilities.

3.2 Fuzzing Process

The fuzzing process is the core of our approach, serving as mechanism for identifying vulnerabilities within the JavaScript engine. Our approach integrates historical information into the weighting system, ensuring that the fuzzing campaigns are not just random but strategically focused. Generating and executing a diverse set of inputs (i.e., JavaScript programs), the fuzzing process systematically probes the engine, targeting areas that historical data has highlighted as particularly vulnerable. The weighting system then dynamically guides these fuzzing campaigns toward the most critical sections of the codebase, optimizing the chances of uncovering hidden flaws that might otherwise go undetected.

3.2.1 Weighting System

A weighting system is a method for assigning importance or relevance to different elements, factors, or solutions in a problem space. It is often used in contexts like machine learning, decision-making, or optimization to prioritize specific options or inputs over others. In decision-making processes, weighting systems help evaluate alternatives by emphasizing critical criteria, ensuring that priorities are reflected accurately in the outcome (Fagin and Wimmers, 2000).

[7]https://msrc.microsoft.com/update-guide/en-US/advisory/CVE-2018-0776

[8]https://msrc.microsoft.com/update-guide/en-US/advisory/CVE-2018-0933

[9]https://msrc.microsoft.com/update-guide/en-US/advisory/CVE-2018-0934

[10]https://github.com

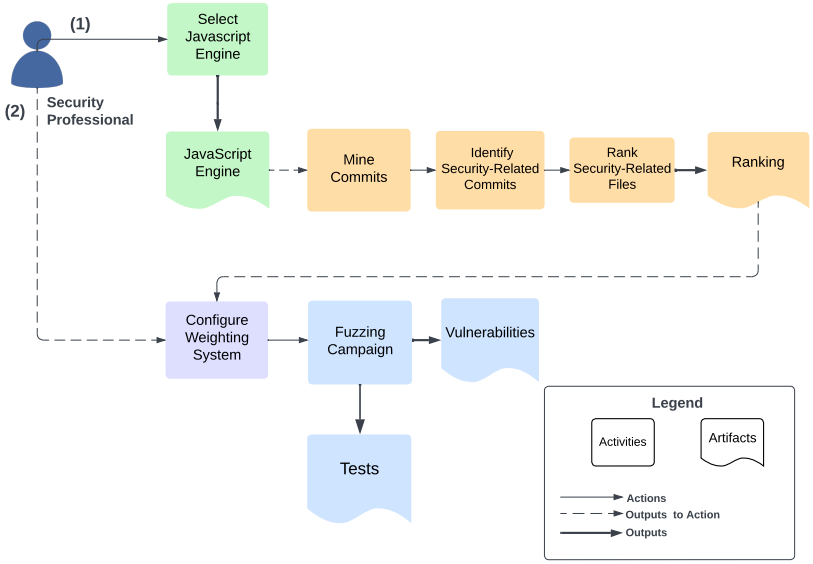


Figure 1: Overview of JSTargetFuzzer.

We integrated a scoring system in JSTargetFuzzer as described below.

- Each input is assessed based on its effectiveness in targeting specific high-risk areas of the code, as identified through historical analysis.
- Higher scores (Y) are assigned to inputs that interact with or impact ranked security-related files.
- Lower scores (X) are given to inputs that do not target the identified areas.
- The user configures the weights for both X and Y, tailoring the scoring system to specific goals.
- Inputs are selected based on their scores, with higher-scoring programs having a greater likelihood of being chosen during the mutation process.

Integrating the weighting system into the approach ensures that the fuzzing process is continuously driven toward exploring the target areas of the code.

3.2.2 Fuzzing Campaign

The weighting system is triggered during the fuzzing campaign in order to target specific address spaces within JavaScript engines. Based on Fuzzilli's original algorithm (Groß et al., 2023), Algorithm 1 shows how the JSTargetFuzzer fuzzing and weighting system operate.

The algorithm receives as input X and Y, which are weights, respectively for relevant programs and programs that specifically hit security-related code. Another input is *targetCov*, a data structure that stores the security-related code areas (Section 3.1). Initially, all programs are generated with weight X and added to the population of relevant programs called Corpus (line 3). Then, the fuzzing loop starts and continues while the fuzzing is enabled (lines 4-24). In line 5, a program P is selected from the corpus based on its weight; so, programs with higher weights are more likely to be selected. The selected program will pass for N mutation iterations (lines 6-23). The selected program is mutated (line 7) and executed in engine under test (line 8). If the mutated program Pm causes a crash, its results are saved to disk, and added to the corpus with weight X (lines 9-11). Otherwise (line 15), it checks if the program is considered interesting; function `isInteresting()`, as in Fuzzilli, is implemented as follows. It returns true if one of the following conditions is met: (i) coverage of newly discovered branches in the JavaScript engine's code; (ii) the program encounters assertion failures, which are indicated by an exit code different from 0 (normal) without crashing. Instead, it produces an output on STDERR, signaling an issue during execution; and (iii) the program times-out during the execution. If at least one of these conditions is satisfied, the program is added to the corpus with weight X. When the program hits new branches, it further checks for security-related coverage in *targetCov* (line 17). If security-related code is hit, the program is added to the corpus with a higher weight Y (line 18).

Algorithm 1: JSTargetFuzzer Fuzzing Process.

```
1: Input: weights X and Y, targetCov
2: Corpus ← []
3: Corpus.add(genSeedProgram(), weight(X))
4: while enabledFuzzing() do
5:     P ← selectElementByWeight(Corpus)
6:     for N iterations do
7:         Pm ← mutate(P)
8:         exec ← execute(Pm)
9:         if exec.returnStatus == crash then
10:            saveToDisk(Pm, exec)
11:            Corpus.add(Pm, weight(X)) {Lower weight X for
               crashing}
12:        else
13:            if exec.returnStatus == normal then
14:                P ← Pm
15:                if isInteresting(exec) then
16:                    Corpus.add(P, weight(X)) {Lower weight
                       X for assertions/timeouts/new branches}
17:                    if NCov in targetCov then
18:                        Corpus.add(P, weight(Y)) {Higher
                           weight for hitting target}
19:                    end if
20:                end if
21:            end if
22:        end if
23:    end for
24: end while
```

4 IMPLEMENTATION

In this section, we outline key implementation details of our approach. This implementation was then used to conduct the experiments.

For the Historical Information part, we adopted scripts to download from GitHub and configure the selected JavaScript engines. To rank the most relevant security-related files for each engine, we reused the experimental package provided by Oliveira et al. (2023). The package provides ML classifiers to identify security-related commits, which we used to compute the most-frequently changed files. We reused the data for ChakraCore and JavaScriptCore engines. Then, we re-executed existing scripts to collect data for engines Duktape and JerryScript.

Concerning Fuzzing Process, we developed JSTargetFuzzer on top of state-of-the-art Fuzzilli (Groß et al., 2023). Fuzzilli is an open-source and extensible tool, enabling us to adapt its structure for our fuzzing process. It is primarily written in Swift and C. To define the address space of the security-related files (*targetCov* in Algorithm 1), we utilized the *GNU Debugger (GDB)* to run the selected JavaScript engine with symbols enabled, allowing us to identify the memory address ranges associated with the files of interest.

To obtain coverage information about the execution of a given program (lines 14-21 in Algorithm 1), we utilized Clang's[11] built-in instrumentation, along with its sanitizer coverage functions. So, the JavaScript engine is compiled with these configurations, providing coverage information for the fuzzing process.

5 EXPERIMENTAL SETUP

In this section, we present the experimental setup adopted to evaluate JSTargetFuzzer. We utilized Fuzzilli as a baseline, as it is a recognized state-of-the-art fuzzing tool integrated into well-known JavaScript engines (Groß et al., 2023). We set out our evaluation to answer the following *Research Questions (RQs)*:

• **RQ1: To what extent is JSTargetFuzzer capable of guiding the fuzzing process to explore security-related files?** This RQ evaluates JSTargetFuzzer's effectiveness in directing its fuzzing efforts toward security-critical areas. This question also examines whether our approach is being properly executed and if there is a significant difference in the concentration of fuzzing efforts and branch discovery between campaigns using JSTargetFuzzer and those using Fuzzilli.

• **RQ2: What are the characteristics of the programs generated with JSTargetFuzzer?** This RQ examines the input generation process, focusing on the structure of the generated programs in terms of operations, parameters, and overall complexity. Our goal is to compare programs produced by JSTargetFuzzer with the ones generated by Fuzzilli, assessing how the weighting influences the characteristics of the programs.

• **RQ3: To what extent is JSTargetFuzzer capable of detecting vulnerabilities?** This RQ aims to analyze whether JSTargetFuzzer can detect vulnerabilities in the engines. If so, we want to verify if Fuzzilli can detect them too.

As subjects, we selected four JavaScript engines based on various criteria: popularity, complexity, security features, and code availability. They are described next:

- **ChakraCore**[12] is the core part of the Chakra JavaScript engine that powers Microsoft Edge and Internet Explorer browsers. It is still a very popular engine with a high market share, making it a valuable target.

[11] https://clang.llvm.org

[12] https://github.com/chakra-core/ChakraCore

- **JavaScriptCore**[13] is the JavaScript engine that forms the backbone of the WebKit framework, which is integral to Apple's Safari browser and a variety of other applications on macOS and iOS.
- **Duktape**[14] is an embeddable JavaScript engine designed for simplicity and low memory usage. Despite its smaller footprint, it remains popular in embedded systems and *Internet of Things (IoT)* devices, presenting unique security challenges.
- **JerryScript**[15] is a lightweight JavaScript engine for IoT devices. It is designed to run on devices with constrained resources, making it a critical component in IoT security research.

For each engine we use the builds shown in Table 1. This table also presents the corresponding numbers of *Lines of Code (LoC)* and source code files. Notice that ChakraCore is the biggest engine with respect to LoC, having more than 2.9 MLoC of C/C++ code. Next is JavaScriptCore, Duktape, and finally JerryScript. We configured each engine with *Address-Sanitizer (ASAN)* to identify issues stemming from improper memory access and utilized debug mode to uncover vulnerabilities related to undefined behavior.

For this study, we opted to identify the Top-1 most frequently modified security-related file from each engine, following the procedure described in Section 4. Table 2 shows the security-related file selected for each engine, along with its LoC. Hence, JSTargetFuzzer would give a greater weight to programs generated during the fuzzing campaigns that hit the address space of these files. In experiments, we set up JSTargetFuzzer using weight $X = 1$ and weight $Y = 1000$. Since only a small percentage of programs can hit new branches in the target address space, we applied a significantly higher value for weight Y to ensure these programs are prioritized during the fuzzing campaigns. The aforementioned decisions are supported by preliminary tests conducted to find balance in the fuzzing process.

We established fuzzing campaigns of 120 minutes, for both JSTargetFuzzer and Fuzzilli, running for the selected engines. Acknowledging the inherent randomness in fuzzing, we executed each campaign three times and averaged results were computed and presented. We employed an Intel(R) i9 14900F CPU (24-cores) computer with 64-bit Kali 2024.1 OS.

In RQ1, we assess the approach's effectiveness by measuring (1) the frequency with which it visits the targeted areas, and (2) its ability to reach unique addresses within the target address space. To capture these two main capacities, we adopt the following metrics:

- **HitCount.** The total number of accesses to the address space of the selected security-related file during fuzzing campaigns. It indicates how frequent branches in the targeted address space are hit.
- **UniqueHitCount.** The number of unique accesses to the address space of the selected security-related file during fuzzing campaigns. It reflects the discovery of new branches within the targeted address space.

As we intend to analyze how these metrics evolve over time, we collected these metrics after each iteration of the fuzzing campaign, associating timestamps.

Concerning RQ2, we analyzed the characteristics of the generated programs. To do so, we modified the fuzzers' implementation so that all programs included in the corpus are persisted. We adopted the following metrics: number of parameters, number of operations, number of loops, and *Cyclomatic Complexity (CC)*. The number of parameters reflects the count of distinct parameters used within the functions, shedding light on the program's data manipulation and potential for variable interactions. The number of operations includes all statements and expressions within the programs, providing an overall measure of program size. Loop operations specifically refer to the frequency of loop constructs, such as for and while loops. The Cyclomatic Complexity refers to the number of linearly independent paths through the program's source code. We implemented a Python script with the Lizard library[16] that computes these metrics from the programs and utilized cloc[17] utility to determine the LoC values.

To address RQ3, we look at potential vulnerabilities in the form of engine crashes or corruption memory issues pointed out by ASAN, observed during the fuzzing campaigns. We also tracked the time taken until those events occurred. For the potential vulnerability, we conducted an in-depth investigation. Initially, we reproduced the issue to confirm the report by the fuzzer. Then, we searched the engine repository for reports about the potential vulnerability, and tested it in the most recent version of the engine.

The raw data, scripts, fuzzers' implementations, and other related artifacts required to replicate this study are available at https://github.com/brunogoliveira-ufpr/JSTargetFuzzer. The experimental package was anonymized due to the double-blind process.

[13] https://github.com/WebKit/WebKit
[14] https://github.com/svaarala/duktape
[15] https://github.com/jerryscript-project/jerryscript
[16] https://github.com/terryyin/lizard
[17] https://github.com/AlDanial/cloc

Table 1: JavaScript engines.

Engine	Build	#LoC	#Files
ChakraCore	c3ead3f8a6e0bb8e32e043adc091c68cba5935e9	2,951,664	820
JavaScriptCore	c6a5bcca33e3147a0aaa5ea1f3aa2384aae383da	698,537	1,190
Duktape	50af773b1b32067170786c2b7c661705ec7425d4	170,256	479
JerryScript	8ba0d1b6ee5a065a42f3b306771ad8e3c0d819bc	102,632	342

Table 2: Top-1 security-related files.

Engine	Top-1 File	#LoC
ChakraCore	GlobOpt.cpp	18,028
JavaScriptCore	JSGlobalObject.cpp	3,092
Duktape	duk_api_stack.c	6,897
JerryScript	ecma-function-object.c	2,093

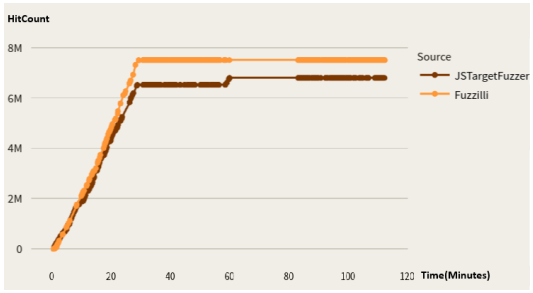
6 ANALYSIS OF RESULTS

In this section we analyse the results in order to answer our RQs.

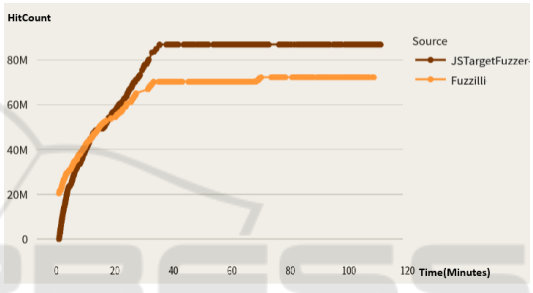
6.1 RQ1: Exploring Security-Related Files

Figure 2 shows how HitCount (y-axis) evolves over the elapsed time (x-axis) in minutes. HitCount grows over time for both approaches, though it increases faster in JSTargetFuzzer. For JavaScriptCore (b), the better performance of JSTargetFuzzer is more noticeable after approximately 30 minutes of fuzzing. By the end, JSTargetFuzzer achieved average HitCounts that were 16.17% higher for JavaScriptCore and 6.83% higher for JerryScript compared to Fuzzilli. For the ChakraCore (a) and Duktape (c) engines, Fuzzilli and JSTargetFuzzer appear to have closer results over time. Nevertheless, JSTargetFuzzer had better HitCount by the end: 3.61% higher for Duktape. In ChakraCore, JSTargetFuzzer achieved results comparable to Fuzzilli, though its performance was, on average, 1.70% lower.

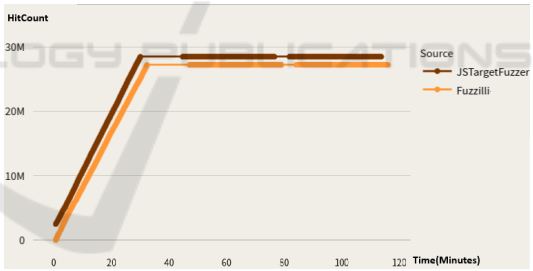
Figure 3 shows how UniqueHitCount (y-axis) evolves over the elapsed time (x-axis) in minutes. For both approaches, UniqueHitCount grows fast for the first 5 minutes, and then grows slowly afterwards. This is expected once there is a large number of uncovered branches at the beginning, and this number is reduced in subsequent iterations of the fuzzing campaigns. Observe that JSTargetFuzzer had a performance better than Fuzzilli after the first minutes. By the end, JSTargetFuzzer was slightly better than Fuzzilli by uncovering more unique branches within the target address space during the fuzzing campaigns, for all JavaScript engines. The greatest difference is in ChakraCore, where JSTargetFuzzer had



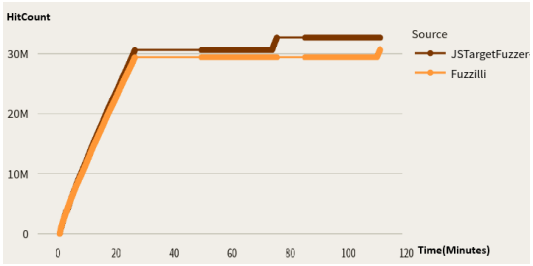
(a) ChakraCore.



(b) JavaScriptCore.



(c) Duktape.

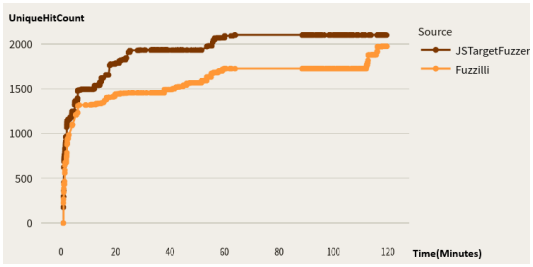


(d) JerryScript.

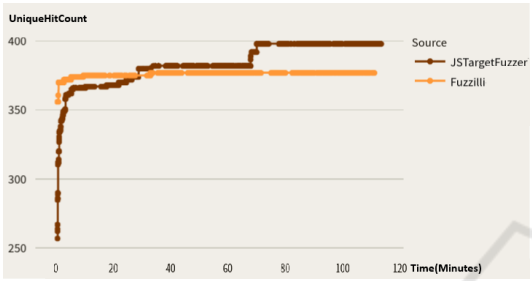
Figure 2: HitCount (y-axis) over time (x-axis).

a UniqueHitCount 15.09% higher than Fuzzilli; this accounts for approximately 306 more branches. Next JavaScriptCore (18 branches – 4.84%), JerryScript (3

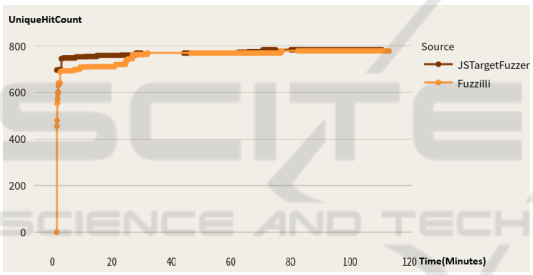
more branches – 1.89%), and Duktape (11 branches – 1.46%).



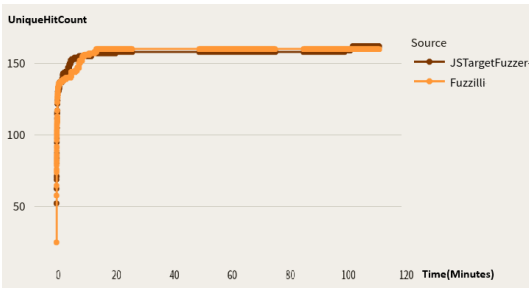
(a) ChakraCore.



(b) JavaScriptCore.



(c) Duktape.



(d) JerryScript.

Figure 3: UniqueHitCount (y-axis) over time in (x-axis).

The particularities of each engine, like modularization, project size, runtime, and optimizations, caused great variations in the metrics. For instance, fuzzing campaigns for JavaScriptCore had up to 80M HitCount, while ChakraCore had campaigns with up to 8M. The number of branches in the security-related file is also a factor. For example, JerryScript had up to 159 explored branches (UniqueHitCount) of 1,256 in total, while JavaScriptCore was limited to around 386 branches of 12,467 in total.

> *Response to RQ1:* JSTargetFuzzer is capable of guiding the fuzzing process to explore more security related areas when compared to Fuzzilli. During the fuzzing campaigns, JSTargetFuzzer hit more branches in security-related files' address space. For HitCount, the improvements varied from 3.61% (Duktape) to 16.17% (JavaScriptCore). While in ChakraCore, Fuzzilli hits in average more than JSTargetFuzzer 1.70%. JSTargetFuzzer also explores more unique branches over time, with improvements from 1.46% (Duktape) to 15.09% (ChakraCore).

6.2 RQ2: Characteristics of Generated Programs

We herein analyzed the corpus of programs generated by JSTargetFuzzer compared with the one generated by Fuzzilli. In addition to this, we also analyze part of this corpus composed of the programs that received the higher weight Y.

JSTargetFuzzer generated an average of 1,371 programs for ChakraCore (X: 1,258, Y: 113), 2,742 for Duktape (X: 2,711, Y: 31), 11,722 for JavaScriptCore (X: 11,685, Y: 37), and 138,713 for JerryScript (X: 272,383, Y: 55). Y-programs constitute a small fraction of the corpus, ranging from 0.02% (JerryScript) to 9.04% (ChakraCore). Despite their limited numbers, Y-programs are more likely to be selected and mutated, significantly influencing the characteristics of programs generated in subsequent iterations.

Table 3 shows the metrics values for the corpus of programs generated by JSTargetFuzzer and Fuzzilli, per engine. For JSTargetFuzzer, we also present the values for programs with weight Y (third column) and the totality of the programs (corpus) (fourth column). One aspect of Fuzzilli (also reflected on JSTargetFuzzer) is that the fuzzing process starts with simple programs that grow with complexity as they are mutated.

In general, the metrics values for the programs of the JSTargetFuzzer's corpus are smaller than the values of the Fuzzilli's corpus. This means that the programs generated by JSTargetFuzzer are simpler and this may imply fast execution during the fuzzing campaign, allowing more iterations in less time. This should be better investigate in future work.

We also observe that the values are smaller for the corpus of Y-programs. Y-programs have smaller mean numbers of parameters (≈22%), operations (≈23%), loops (≈20%), and cyclomatic complexity (≈21%). This may have occurred because most Y-programs are added to the corpus earlier in the fuzzing campaigns, as branches in the target code area become harder to cover. Only programs that can hit new addresses in the target space will be given a higher weight.

A different behavior was noticed in the mean CC values of JavaScriptScore. The values of Fuzzilli are lower than the ones of JSTargetFuzzer. In JavaScriptCore, JSTargetFuzzer generates fewer loops; however, these loops often feature deeper nesting or include additional branching logic within their bodies, significantly increasing CC. This behavior is likely driven by the weighting system, which prioritizes programs that target specific, security-related branches in the code. In this instance, JSTargetFuzzer generates programs with more complex control flow structures to effectively trigger the target address space, resulting in more independent execution paths and, consequently, increased cyclomatic complexity.

Table 3: Metric values for RQ2, per engine.

Metric	Engine	JSTargetFuzzer		Fuzzilli
		Weight Y	Corpus	
# parameters	ChakraCore	1.76	2.47	2.98
	JavaScriptCore	3.68	4.13	3.11
	JerryScript	3.61	4.35	5.76
	Duktape	4.04	4.60	7.10
# operations	ChakraCore	40.96	58.33	71.98
	JavaScriptCore	95.35	107.41	76.59
	JerryScript	78.56	98.83	141.81
	Duktape	85.40	94.96	180.05
# loops	ChakraCore	0.49	0.45	0.52
	JavaScriptCore	0.42	0.57	0.54
	JerryScript	0.32	0.48	0.63
	Duktape	0.52	0.61	0.98
CC	ChakraCore	1.90	2.59	3.42
	JavaScriptCore	4.50	4.78	3.45
	JerryScript	3.55	4.36	5.88
	Duktape	4.18	4.81	8.80

Response to RQ2: JSTargetFuzzer generates programs with characteristics distinct from Fuzzilli, with Y-programs generally exhibiting lower metric values overall.

6.3 RQ3: Uncovered Vulnerabilities

No vulnerability was detected in ChakraCore, JavaScriptCore, and Duktape; this result was expected once these engines were widely used and tested in practice. On the other hand, JSTargetFuzzer revealed a vulnerability in JerryScript. In comparison, Fuzzilli could not uncover any vulnerability within the same time frame.

The vulnerability found in JerryScript is a stack-overflow, caused by infinite recursion within the engine. This occurs when a function repeatedly calls itself (directly or indirectly) without a proper exit condition, eventually consuming all available stack memory. In this case, the recursion seems to involve two functions, which repeatedly call each other until the stack is exhausted. This issue is no longer present in the latest version we tested[18]. We observed that JSTargetFuzzer uncovered the vulnerability in all three repetitions of the fuzzing campaigns and took an average of 23 minutes to detect it.

Response to RQ3: JSTargetFuzzer detected a vulnerability in JerryScript. For the same time frame, Fuzzilli did not uncover any vulnerability.

7 DISCUSSION

In this section, we discuss some limitations observed during our study, as well as, possible threats to the validity of our results.

7.1 Threats to Validity

Although we attempted to mitigate them to the best of our ability, this work contains some threats to its validity, as follows.

Internal Validity: The fuzzers employed in the experiments took a lot of random choices, so its impact introduces a threat. To mitigate this, all results were based on mean values of three executions. However, more executions would be better to deal with randomness. A period of two hours was used for the fuzzing campaigns. We decided to adopt this short period because time reduction is a motivation for proposing our approach. However, this may lead to incomplete coverage in engines that require more time for effective exploration. Consequently, the results might not accurately reflect the comparative effectiveness of the

[18]https://github.com/jerryscript-project/jerryscript/commit/2dbb6f7

tools over extended testing periods. Another threat is related to possible implementation errors in our code to identify the security-related files, and collect the metrics. Moreover, the values for the weights X and Y were obtained empirically. Other configurations should be better explored in future work.

Conclusion Validity: Our findings depend on the used indicators and metrics adopted in the analysis of the results. The use of other indicators may lead to different results. To minimize this threat, we make the experimental package available for future replication.

External Validity: We analyze only four engines. Therefore, it is not possible to generalize our results. Our engines should be considered in a future experiment.

7.2 Limitations

In our study we observed some points that need to be deeply studied in future research, or to be considered for practical use of JSTargetFuzzer. The effectiveness of the weighting system used to prioritize programs during the fuzzing process is heavily influenced by the number of programs generated during the fuzzing campaign. Specifically, for smaller JavaScript engines like Duktape and JerryScript, which tend to generate more programs within a given time frame due to their compact size, the predetermined weight values might require significant adjustment to maintain efficacy.

If the weights are set too low, the fuzzer might not sufficiently prioritize the programs targeting critical areas, potentially missing vulnerabilities. Conversely, if the weights are set too high, the fuzzer could overprioritize certain programs, leading to an inefficient allocation of resources and possibly overlooking other critical parts of the engine.

Focusing specific parts of the code may lead to overlook vulnerabilities in other parts of the JavaScript engine. JSTargetFuzzer may miss out on discovering security issues outside these targeted regions while concentrating on specific areas identified through historical data. This narrowed focus could lead to an incomplete assessment of the engine's overall security posture. However, it's important to note that JSTargetFuzzer still incorporates a comprehensive fuzzing approach by allowing other programs to interact with the engine. This broader interaction helps ensure that while the primary focus is on critical areas, the rest of the engine is not neglected, maintaining a balance between targeted and general fuzzing efforts.

Smaller engines like JerryScript and Duktape, which have less overhead, may allow for faster execution of fuzzing campaigns, potentially leading to quicker detection of vulnerabilities or more efficient coverage metrics. This variability across different engines could skew the results, making JSTargetFuzzer appear more effective or efficient than it might be in other contexts, particularly in larger or more complex engines.

8 CONCLUDING REMARKS

This paper introduces JSTargetFuzzer, an approach that incorporates a novel weighting system designed to target specific areas of interest during fuzzing campaigns using historical information. These areas are identified by mining security information from the commit history. Our evaluation revealed that JSTargetFuzzer directs fuzzing campaigns toward specified security-related code areas. This capability enables the fuzzer to concentrate on particular areas of the JavaScript engine, increasing the likelihood of uncovering vulnerabilities in those regions. JSTargetFuzzer found one vulnerability in JerryScript, which was missed by the baseline fuzzer. This is particularly significant given that two hours is a very limited time to detect vulnerabilities, highlighting a promising efficiency.

As a limitation, JSTargetFuzzer may miss out on discovering security issues outside these targeted regions. To reduce this limitation, we intend to investigate strategies based on historical information, encompassing every stage of fuzzer development, including seed generation, mutation operators, and integration of our weighting system, along with considering different oracles. The seeds, mutation operators, and oracles should be tailored to specific types of vulnerabilities. The idea is to carefully consider the characteristics of these vulnerabilities when creating initial seeds and designing mutation operators that align with the vulnerability patterns. In some cases, we will define oracles capable of detecting anomalous behaviors to identify security issues more effectively.

In future, one potential direction is to explore different historical information so that other aspects of the JavaScript engines are considered like robustness, performance, and so on. Furthermore, the idea of using historical information to leverage fuzzing could be applied to other kinds of software systems.

ACKNOWLEDGMENTS

Andre T. Endo is partially supported by grant #2023/00577-8, São Paulo Research Foundation

(FAPESP); Silvia Regina Vergilio is supported by grant #310034/2022-1, CNPq. This work was also supported by Coordination for the Improvement of Higher Education Personnel (CAPES) - Program of Academic Excellence (PROEX).

REFERENCES

Bernhard, L., Scharnowski, T., Schlögel, M., Blazytko, T., and Holz, T. (2022). JIT-Picking: Differential fuzzing of JavaScript engines. In *ACM Conference on Computer and Communications Security CCS*, pages 351–364. ACM.

Eom, J., Jeong, S., and Kwon, T. (2024). Fuzzing JavaScript interpreters with coverage-guided reinforcement learning for LLM-based mutation. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24)*, pages 1–13, New York, NY, USA. ACM.

Fagin, R. and Wimmers, E. L. (2000). A formula for incorporating weights into scoring rules. *Theoretical Computer Science*, 239(2):309–338.

Groß, S., Koch, S., Bernhard, L., Holz, T., and Johns, M. (2023). Fuzzilli: Fuzzing for JavaScript JIT compiler vulnerabilities. In *Network and Distributed Systems Security (NDSS) Symposium 2023*, pages 10–25, San Diego, CA, USA.

Han, H., Oh, D., and Cha, S. (2019). CodeAlchemist: Semantics-aware code generation to find vulnerabilities in JavaScript engines. In *Network and Distributed System Security Symposium*.

He, X., Xie, X., Li, Y., Sun, J., Li, F., Zou, W., Liu, Y., Yu, L., Zhou, J., Shi, W., et al. (2021). Sofi: Reflection-augmented fuzzing for JavaScript engines. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2229–2242.

Holler, C., Herzig, K., and Zeller, A. (2012). Fuzzing with code fragments. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 445–458, Bellevue, WA. USENIX Association.

Kang, Z. (2021). A review on JavaScript engine vulnerability mining. In *Journal of Physics: Conference Series*, volume 1744, page 042197. IOP Publishing.

Kienle, H. M. (2010). It's about time to take JavaScript (more) seriously. *IEEE Software*, 27(3):60–62.

Lee, S., Han, H., Cha, S. K., and Son, S. (2020). Montage: A neural network language Model-Guided JavaScript engine fuzzer. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2613–2630. USENIX Association.

Li, F. and Paxson, V. (2017). A large-scale empirical study of security patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2201–2215, New York, NY, USA. Association for Computing Machinery.

Mozilla (2022). jsfunfuzz. https://github.com/MozillaSecurity/funfuzz. Accessed in 08/24/2022.

Oliveira, B. G., Endo, A. T., and Vergilio, S. (2023). Characterizing security-related commits of JavaScript engines. In *In Proceedings of the 25th International Conference on Enterprise Information Systems (ICEIS)*, volume 2, pages 86–97.

Park, S., Xu, W., Yun, I., Jang, D., and Kim, T. (2020). Fuzzing JavaScript engines with aspect-preserving mutation. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1629–1642.

Shin, Y. and Williams, L. (2008). Is complexity really the enemy of software security? In *Proceedings of the 4th ACM Workshop on Quality of Protection*, QoP '08, page 47–50, New York, NY, USA. Association for Computing Machinery.

Sun, L., Wu, C., Wang, Z., Kang, Y., and Tang, B. (2022). KOP-Fuzzer: A key-operation-based fuzzer for type confusion bugs in JavaScript engines. In *2022 IEEE 46th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 757–766.

Tian, Y., Qin, X., and Gan, S. (2021). Research on fuzzing technology for JavaScript Engines. In *Proceedings of the 5th International Conference on Computer Science and Application Engineering*, pages 1–7.

Wang, J., Chen, B., Wei, L., and Liu, Y. (2019). Superion: Grammar-aware greybox fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 724–735.

Wang, J., Zhang, Z., Xin, Q. A., Liu, S., Du, X., and Chen, J. (2023). FuzzJIT: Oracle-Enhanced fuzzing for JavaScript engine JIT compiler. In *32nd USENIX Security Symposium (USENIX Security 23)*, Anaheim, CA. USENIX Association.

Xiang, Y., Zhang, X., Liu, P., Ji, S., Liang, H., Xu, J., and Wang, W. (2024). Critical code guided directed greybox fuzzing for commits. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 2459–2474, Philadelphia, PA. USENIX Association.

Zhu, X. and Böhme, M. (2021). Regression greybox fuzzing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 2169–2182, New York, NY, USA. Association for Computing Machinery.